1995

Linear-Time Algorithms for Parametric Minimum Spanning Tree Problems on Planar Graphs

David Fernández-Baca
Iowa State University, fernande@iastate.edu

Giora Slutzki
Iowa State University, slutzki@iastate.edu

Linear-Time Algorithms for Parametric Minimum Spanning Tree Problems on Planar Graphs

Abstract
A linear-time algorithm for the minimum-ratio spanning tree problem on planar graphs is presented. The algorithm is based on a new planar minimum spanning tree algorithm. The approach extends to other parametric minimum spanning tree problems on planar graphs and to other families of graphs having small separators.

Disciplines
Theory and Algorithms

Linear-Time Algorithms for Parametric Minimum Spanning Tree Problems on Planar Graphs*

David Fernández-Baca† and Giora Slutzki
Department of Computer Science, Iowa State University, Ames, IA 50011

Abstract

A linear-time algorithm for the minimum-ratio spanning tree problem on planar graphs is presented. The algorithm is based on a new planar minimum spanning tree algorithm. The approach extends to other parametric minimum spanning tree problems on planar graphs and to other families of graphs having small separators.

1 Introduction

Suppose we are given an undirected graph G where each edge e has two weights a_e and b_e; the b_e's are assumed to be either all negative or all positive. The *minimum ratio spanning tree problem* (MRST) [Cha77] is to find a spanning tree T of G such that the ratio $\sum_{e\in T} a_e / \sum_{e\in T} b_e$ is minimized. One application of MRST arises in the design of communication networks. The number a_e represents the cost of building link e, while b_e represents the time required to build that link. The goal is to find a tree that minimizes the ratio of total cost over construction time. Other applications of MRST are given elsewhere [CMV89, Meg83]. The main result of this paper is a linear-time algorithm for solving parametric minimum spanning tree problems on

*A preliminary version of this paper will appear in the Proceedings of the *2nd Latin American Theoretical Informatics Conference*, Viña del Mar, Chile, 1995.

†Supported in part by the National Science Foundation under grant No. CCR-211262.

planar graphs and other families of graphs with small separators. The approach leads to linear-time planar MRST algorithm, as well as to linear-time algorithms for sensitivity analysis and for Lagrangian relaxation problems associated with minimum spanning trees. To achieve our results we have developed a new linear-time planar minimum spanning tree algorithm based on graph decomposition and graph reduction.

The best known MRST algorithm for arbitrary graphs, due to Cole [Cole87], is an application of Megiddo's method of parametric search [Meg79, Meg83]. Like other algorithms for the problem (including ours), Cole's relies on the equivalence between MRST and the following parametric search problem [Cha77]. Associate with each edge $e \in G$ a linear weight function $w_e(\lambda) = a_e - \lambda b_e$ and let $Z(\lambda)$ denote the weight of the minimum spanning tree relative to the weights $w_e(\lambda)$. The problem is to find the root λ^* of $Z(\lambda)$. Cole's method determines a minimum ratio spanning tree in $O(T_{MST}(n,m)\log n)$ time, where $T_{MST}(n,m)$ denotes the time to compute a minimum spanning tree of an n-vertex, m-edge graph. The best deterministic minimum spanning tree algorithm achieves $T_{MST}(n,m) = O(m \log \beta(m,n))$ [GGST86], resulting in a $O(m \log \beta(m,n) \log n)$ for the general minimum ratio spanning tree problem. Faster MRST algorithms can be obtained either by using Karger, Klein, and Tarjan's $O(m)$ randomized minimum spanning tree algorithm [KKT94], or Fredman and Willard's deterministic $O(m)$-time minimum spanning tree algorithm, which operates under a less restrictive model of computation [FrWi90]. For planar graphs, a minimum spanning tree can be constructed in $O(n)$ time [ChTa76] (see also Section 3), leading to a $O(n \log n)$ MRST algorithm.

Parametric search has been the subject of a considerable amount of research in recent times because of its numerous applications to optimization and computational geometry [CoMe93, Tol93a, CEGS92, MaSc93]. In the context of optimization problems such as MRST, the application of Megiddo's technique tends to follow a common pattern. Suppose we have an algorithm A that allows us to determine the value of a certain function f for any λ within a certain range (for MRST, algorithm A would be any minimum spanning tree algorithm), and that we wish to locate a critical value λ^* for f. To find λ^*, we simulate the execution of A to determine its computation path at λ^*. In the simulation, the operations of A are executed symbolically, manipulating functions of λ instead of numbers; this is referred to as *lifting* the computation of A. To determine the outcome of comparisons without ex-

plicit knowledge of λ^*, the simulation invokes an oracle, which is often closely related to A itself. Since oracle calls are expensive, they must be used sparingly. Megiddo showed that if these operations can be batched (i.e., grouped and ordered in such a way as to permit many of them to be resolved by a single oracle call), the total amount of work to solve the parametric problem can often be made at most a polylogarithmic factor slower than that of algorithm A. The polylogarithmic slowdown in going from non-parametric to parametric algorithms remains even when using Cole's technique [Cole87]. The slowdown is largely a consequence of treating the oracle as a black box. Frederickson [Fre90] observed that, as the search progresses, it is sometimes possible to compile information that can speed up subsequent oracle calls and used this idea to devise linear-time algorithms for a variety of location problems on trees. Subsequently it was shown that a large class of parametric optimization problems can be solved in linear time for graphs of bounded tree-width [FeSl94].

Our MRST algorithm is influenced by Frederickson's work. It is a departure from Megiddo's algorithm for general graphs in that it does not use sorting at a global level, and hence does not need to depend on the AKS sorting network [AKS83], whose large constants of proportionality make it impractical. Our algorithm simulates a new (non-parametric) minimum spanning tree algorithm that takes advantage of planarity to view the input graph at different levels of refinement (a technique similar to that used by Frederickson for computing shortest paths [Fre85b]). The structure of the non-parametric algorithm allows us to construct a sequence of successively faster oracles as the simulation unfolds. In the process, we will identify and contract an increasingly larger set of *essential edges*; i.e., edges that must be included in any minimum spanning tree at λ^*. At the conclusion, the input graph will be contracted to a single vertex and λ^* will be found by exhaustive enumeration.

Organization of the paper. Section 2 defines the notion of a multilevel division of a planar graph and describes how to compute one in linear time. Multilevel divisions, together with graph reduction, are used in the linear-time planar minimum spanning tree algorithm presented in Section 3. This algorithm will be the basis for the parametric search scheme discussed in Section 4. The same idea can be used for other parametric spanning tree problems, as discussed in Section 5.

2 Multilevel divisions of planar graphs

Our non-parametric planar minimum spanning tree algorithm relies on an idea by Frederickson [Fre85b], who described an algorithm that uses Lipton and Tarjan's planar separator theorem [LiTa79] to build a division of a planar graph G into *regions*. Each region has two types of vertices, *boundary* vertices and *interior* vertices. Every interior vertex is contained in exactly one region and is adjacent only to vertices within that region. Boundary vertices are shared between at least two regions. Frederickson [Fre85b] showed that, for every positive integer r, G has an *r-division*, i.e., a division with $\Theta(n/r)$ regions of $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices each.

Suppose we are given integers $r_1 > r_2 > \ldots > r_k$, where $r_1 \leq n$ and $r_k \geq 1$. A *multilevel division* of G is constructed as follows. First form an r_1-division of G; each of the resulting regions will be referred to as an *r_1-region*. Now we do the next step for $i = 1, \ldots, k-1$. Take every r_i-region and construct an r_{i+1}-division for it; each resulting region will be referred to as an *r_{i+1}-region*. Note that in this construction, every boundary vertex in an r_i-region A will be considered a boundary vertex for any subregion within A that contains it. It is straightforward to verify that there are $O(r_i/r_{i+1})$ r_{i+1}-regions within every r_i-region and that the total number of r_i-regions is $O(n/r_i)$ [Fre85b].

Lemma 2.1 *For any given integers $r_1 > r_2 > \ldots > r_k$, where $r_1 \leq n$ and $r_k \geq 1$, a* multilevel division *of G can be constructed in $O(n+n\sum_{i=1}^{k} \log r_i/\sqrt{r_i})$ time. If we choose $r_k = \beta$ and $r_i = \beta r_{i+1}$, $1 \leq i \leq k-1$, for some constant $\beta > 1$, the total time is $O(n)$.*

Note: A similar result is claimed by Klein et al. [KRRS94]; we include a proof for completeness.

Proof. We use a result by Goodrich [Goo93] to show that after a one-time-only $O(n)$ preprocessing step, an r-division can be constructed in $O(n \log r/\sqrt{r})$ time. The r-division is built using a two-step procedure by Frederickson [Fre85b] that uses the planar separator theorem [LiTa79]. The latter states that if G has vertex weights adding up to at most one, then there is a partition of $V(G)$ into sets A, B, and C such that no edge joins a vertex of A

with a vertex in B, neither A nor B has total weight exceeding $2/3$, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices.

Frederickson's algorithm starts with G consisting of one region and with all vertices interior. In the first step, it applies the separator theorem with all vertex weights equal to $1/n$, to obtain sets A, B, and C. Two regions with vertex sets $A \cup C$ and $B \cup C$ are inferred, each of which has C as its set of boundary vertices. The same procedure is applied recursively to any region with more than r vertices, resulting in a division of G into $\Theta(n/r)$ regions with no more than r vertices each and a total of $O(n/\sqrt{r})$ boundary vertices [Fre85b]. In the second step, the following operation is repeated until it no longer applies. Find a planar separator in any region R with more than $\sqrt{r}$ boundary vertices and use it to split R into subregions. For this, let the n' boundary vertices have weight $1/n'$, and let interior vertices have weight zero.

The key to implementing Frederickson's algorithm efficiently is to find the required separators quickly. Given $O(n)$ preprocessing time, an algorithm by Goodrich [Goo93] builds a data structure that allows one to compute a separator (weighted or not) in $O(\sqrt{n}\log n)$ time. Within the same time bound, similar data structures can be set up for each region defined by the separator [Goo93]. We should note that Goodrich's algorithm is not directly applicable to our needs, since, after finding the partition of $V(G)$ into sets A, B, and C, it builds data structures for computing separators in $G[A]$ and $G[B]$, and not in $G[A \cup C]$, $G[B \cup C]$ as required by Frederickson's algorithm[1]. Fortunately, only slight changes to Goodrich's scheme are needed to handle this — we omit the details.

Thus, after the one-time-only $O(n)$ preprocessing step needed to build the necessary data structures, we will be able to compute each separator in sublinear time. Hence, the total time for the first step of Frederickson's algorithm is described by the recurrence

$$t(n,r) \le a\sqrt{n}\log n + t(\alpha n + b\sqrt{n}, r) + t((1-\alpha)n + b\sqrt{n}, r) \quad \text{for } n > r,$$

$$t(n,r) = 0 \quad \text{for } n \le r$$

where $1/3 \le \alpha \le 2/3$. One can show by induction that

$$t(n,r) \le cn\log r/\sqrt{r} - d\sqrt{n}\log n$$

[1] We write $G[A]$ denote the subgraph of G induced by the vertices of A.

for some constants c and d. At the beginning of the second step, we have $\Theta(n/r)$ regions of at most r vertices each and with a total of $O(n/\sqrt{r})$ boundary vertices. At the end of the second step, we still have $\Theta(n/r)$ regions of at most r vertices, but now each region has $O(\sqrt{r})$ boundary vertices. The total number of separator computations needed to go from the set of regions existing after the first step to the set of regions at the end of step 2 is therefore $O(n/r)$. For each such region, we have a data structure that allows us to find separators (with the appropriate weights) in $O(\sqrt{r}\log r)$ time. Thus, the total time spent on the second step will be $O(n\log r/\sqrt{r})$, which is asymptotically equal to the time spent on the first step.

The lemma follows by adding up the work for constructing r_i-regions over all i. □

3 Spanning trees via graph reduction

An important consideration in Megiddo's parametric search method is choosing the right *non-parametric* algorithm to simulate. In the context of MRST, we need an algorithm that evaluates $Z(\lambda)$ for any fixed λ; i.e., an algorithm for finding minimum spanning trees in planar graphs. As we stated earlier, Cheriton and Tarjan have devised a $O(n)$ time algorithm for this purpose [ChTa76]; unfortunately, it is not clear how to use it directly to devise an efficient MRST procedure. In this section, we give a new linear-time (non-parametric) minimum spanning tree algorithm for planar graphs that relies on multilevel divisions and the idea of *graph reduction*. While our algorithm is asymptotically no faster than Cheriton and Tarjan's, the way in which it discards larger and larger sets of edges from the graph as the computation unfolds will be a notable advantage from the point of view of parametric search.

3.1 Graph reduction

Our non-parametric minimum spanning tree algorithm works by repeatedly reducing regions of the graph, replacing them by smaller "substitutes" that retain all the essential information for computing minimum spanning trees. The basic procedure for this is algorithm REDUCE, which takes a region A of G having B as its boundary vertices and, via a series of deletions and

contractions of edges, reduces $G[A]$ to a graph with $O(|B|)$ vertices. REDUCE also returns the total cost C of the edges it contracts, these being the edges that participate in every minimum spanning tree of G [2]. The procedure is essentially the same as an algorithm by Lengauer [Len87]; in its description, we will write that an edge e is *contractible* if

- it has a degree-one endpoint that is not a boundary vertex, or
- it shares a degree-two non-boundary vertex with another edge f such that $\text{cost}(e) \leq \text{cost}(f)$.

```
REDUCE(A, C)
1   compute a minimum spanning forest T_A of G[A]
2   discard all edges in E(G[A]) − E(T_A)
3   discard all isolated vertices from G[A]
4   C ← 0
4   while there is a contractible edge in G[A] do
5       choose a contractible edge e
6       C ← C+ cost(e)
7       contract e
```

Observe that step 1 of REDUCE computes a minimum spanning *forest* of region A, rather than a tree. The reason for this is that, even if G is connected, $G[A]$ may not be. The application of REDUCE to a planar graph is shown in Figure 1. The next two lemmas state some properties of REDUCE. We will refer to the graph resulting from applying REDUCE to a graph G or to a subregion A of G as a *reduction* of G.

Lemma 3.1 *Let A be an r-vertex region of G with a set of boundary vertices B. Then, the reduction of $G[A]$ is a graph of size $O(|B|)$. If G is planar and the time for computing a minimum spanning tree is excluded,* REDUCE(A, C) *takes $O(r)$ time and the reduction of G is a planar graph.*

[2] This assumes that the minimum spanning tree is unique, which is guaranteed to be the case if all edge costs are distinct. If they are not, one standard way of ensuring uniqueness is to assign an arbitrary numbering to edges and to use it to resolve ties.

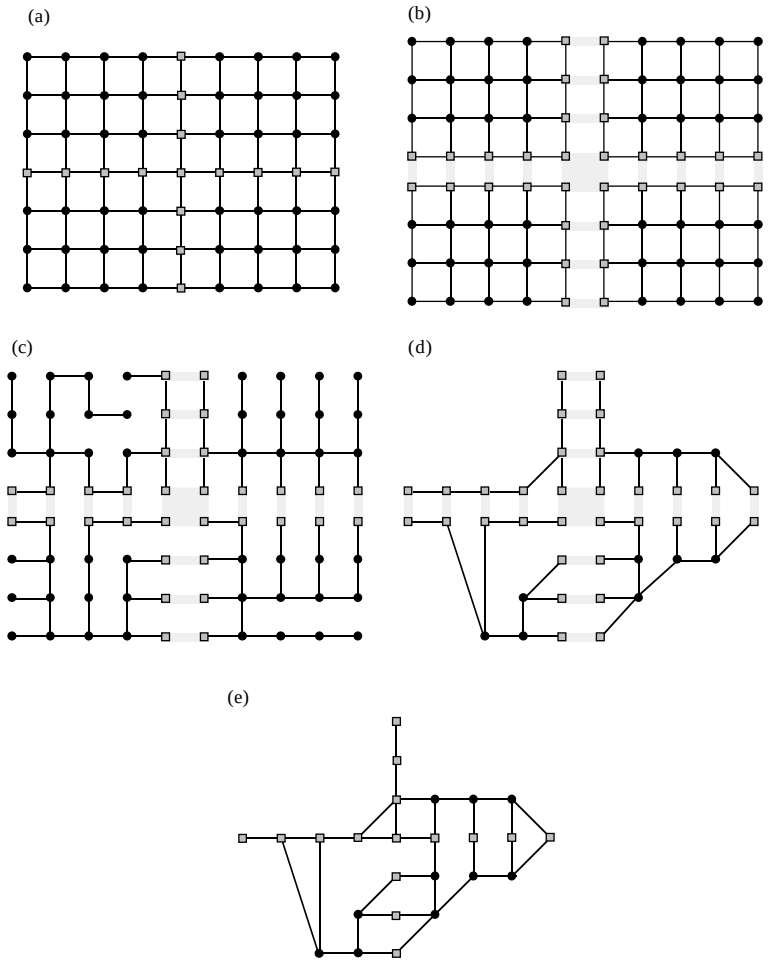

Figure 1: Applying REDUCE to the regions of a graph. (a) The original graph; terminal vertices are shown as grey squares. (b) Separating the graph into four regions. (b) The regions after edges not in their respective minimum spanning forests are deleted. (c) The regions after edge contractions. (d) Reassembling the regions.

Proof. The planarity of the reduction of G follows from the fact that edge deletion and edge contaction are planarity-preserving operations. The size bound on the reduction of $G[A]$ and the fact that REDUCE's deletions and contractions can be done in time linear in the number of vertices and edges were proved by Lengauer [Len87]. The running time for planar graphs is $O(r)$ since the number of edges in these graphs is linear in the number of vertices. □

Note: If the Cheriton-Tarjan planar minimum spanning tree algorithm is used in step 1 of REDUCE, this algorithm will take a total of $O(r)$ time. We will show, however, that a linear-time minimum spanning tree algorithm can be obtained even when a slower minimum spanning tree algorithm (e.g., Kruskal's [Kru56, Tar83]) is used in that step.

Lemma 3.2 *Let G be a graph and let G' be the graph obtained from G by calling* REDUCE(A, C)*, for some region A of G. Then, every edge in the minimum spanning tree T' of G' is in a minimum spanning tree of G. Moreover, the cost of a minimum spanning tree for G equals cost$(T') + C$.*

Proof. Using well-known properties of minimum spanning trees [Len87, Tar83], one can show that every contractible edge is essential, as it is included in every minimum spanning tree of G, and that every discarded edge is non-essential in that it participates in no minimum spanning tree of the graph. The lemma follows. □

3.2 The minimum spanning tree algorithm

We proceed to describe the algorithm MST, which returns the cost of a minimum spanning tree. The procedure will associate with each region A, a variable C_A, which will record the total cost of contracted edges within A. For convenience, we define $r_{k+1} = 1$ and consider every node in G as an r_{k+1}-region. We assume that $C_A = 0$ for every r_{k+1}-region.

```
MST(G)
1   build a multilevel division of G
2   for i ← k downto 1 do
3       for each r_i-region A do
```

4 REDUCE(A, C_A)
5 $C_A \leftarrow C_A + \sum\{C_B : B$ is an r_{i+1}-subregion of $A\}$
6 construct a minimum spanning tree T' of the remaining graph
7 **return** cost$(T') + \sum\{C_A : A$ an r_1-region$\}$

We shall refer to the iteration of MST where $i = j$ as *iteration* j; thus, iteration k is actually the first iteration and iteration 1 is the last. We first state a simple bound on the size of an r_i-region at iteration i of MST.

Lemma 3.3 *At the beginning of iteration i, $k \geq i \geq 1$ of* MST*, every r_i-region of G has been reduced to a planar graph of size $O(r_i/\sqrt{r_{i+1}})$.*

Proof. Each r_i-region A has $O(r_i/r_{i+1})$ r_{i+1}-subregions, each of which has $O(\sqrt{r_{i+1}})$ boundary vertices. By Lemma 3.1, at the beginning of iteration i, each subregion has been reduced to a planar graph with $O(\sqrt{r_{i+1}})$ vertices. The lemma follows. □

Lemma 3.4 *Algorithm* MST *correctly computes the cost of a minimum spanning tree of G in $O(n + n\sum_{i=1}^{k} \log r_i/\sqrt{r_{i+1}})$ time. Suppose that (i) the minimum spanning forests in step 1 of* REDUCE *and step 6 of* MST *are computed using an algorithm that runs in $O(m_0 \log n_0)$ time, where n_0 and m_0 are the number of vertices and edges of input region, and (ii) we choose k and a sequence $r_1, \cdots, r_k$ such that $r_k = \beta$, $r_i = \beta r_{i+1}$, $1 \leq i \leq k-1$, and $\log^2 n \leq r_1 \leq n$, for some suitable constant $\beta > 1$. Then* MST *runs in $O(n)$ time.*

Proof. The correctness of MST follows immediately from Lemma 3.2. By Lemma 2.1, step 1 takes $O(n + n\sum_{i=1}^{k} \log r_i/\sqrt{r_i})$ time. Now, consider steps 2–5. By Lemma 3.3, each region A considered in steps 3–4 has $O(r_i/\sqrt{r_{i+1}})$ vertices and edges. Thus, using Lemma 3.1 and implementing REDUCE's minimum spanning tree computation using a $O(m_0 \log n_0)$ algorithm, we can process A in $O((r_i/\sqrt{r_{i+1}}) \log r_i)$ time, for a total of $O(n \log r_i/\sqrt{r_{i+1}})$ over all r_i-regions. The graph remaining in step 6 will have $O(n/\sqrt{r_1})$ vertices. Since $r_1 \geq \log^2 n$, its minimum spanning tree can be constructed in $O(n)$ time. The total time spent in steps 1–6 is thus $O(n + n\sum_{i=1}^{k} \log r_i/\sqrt{r_{i+1}})$, which is $O(n)$ for the given choice of r_i's. □

Later in this paper, we will find it convenient to choose a sequence $r_1, \ldots, r_k$ such that $r_1 = n$. If we do so, the only r_1-region will be G itself and it will have no boundary vertices. The last iteration of MST will therefore produce a one-vertex graph and C_G will equal the cost of a minimum spanning tree.

4 The search

As stated in the introduction, solving the minimum ratio spanning tree problem is equivalent to finding a value λ^* such that $Z(\lambda^*) = 0$. Our MRST algorithm accomplishes this by lifting the execution of algorithm MST of Section 3 so as to determine all its computation paths over an interval $\mathcal{I}$ containing λ^*. The final result will be a complete description of $Z(\lambda)$ within $\mathcal{I}$; λ^* will be found by searching this description to locate the point at which Z crosses the λ-axis. Lifting is an expensive operation, since the number of computation paths grows rapidly with the size of $\mathcal{I}$. Thus, in order to make the search efficient, we must control the size of this interval; this is accomplished using an *oracle*, a procedure that can determine the position of any given value λ_0 with respect to λ^* [Meg79, Meg83]. One of the key features of our algorithm is the way in which the oracle and the lifting steps interact. Taking advantage of the structure of algorithm MST, we are able to ensure that each successive lifting step uses a faster oracle than the preceding step; the overall effect will be that the time for successive steps adds up in a geometric series.

We will now give an overview of the search algorithm, after which we will describe its main details.

4.1 An overview of the algorithm

The search algorithm lifts the computation of MST iteration by iteration; each lifting step produces an *efficient representation* of all possible outcomes of the ith iteration of MST over some interval $\mathcal{I}$, for each r_i-region A of G. An efficient representation is a data structure that allows us to obtain two objects quickly:

- the reduced graph $G_A(\lambda)$ for A relative to the weights $w_e(\lambda)$ and

- the cost $C_A(\lambda)$ of the contracted edges for that region.

In order to lift iteration i, we will need efficient representations of the outcome of iteration $i+1$ for every r_{i+1}-region B of G. Underlying the representation for region B is a set of values $L_B \subset \mathcal{I}$, which induces a subdivision of $\mathcal{I}$ into a sequence of intervals, denoted $\mathcal{I}_B$. The L_B's will satisfy the following properties.

(P1) Every value in L_B is the λ-coordinate of the intersection point of the costs lines for two edges lying within region B.

(P2) Let $\mathcal{I}'$ be any subinterval of $\mathcal{I}_B$. Then, the computation path followed by MST on B up to the beginning of iteration i is the same for all $\lambda \in \mathcal{I}'$. That is, at the beginning of iteration i, MST will have deleted and contracted exactly the same set of edges in region B, for all $\lambda \in \mathcal{I}'$.

(P3) $|\bigcup\{L_B : B \text{ an } r_{i+1}\text{-region}\}| \leq n/r_{i+1}$.

By (P1), $|L_B| = O(r_{i+1}^2)$, since region B has $O(r_{i+1})$ edges. By (P2), the reduced graph $G_B(\lambda)$ is the same for all $\lambda \in \mathcal{I}'$ and that $C_B(\lambda)$ is a straight line within $\mathcal{I}'$. Thus, the outcomes of all computation paths on B up to the beginning of iteration i can be represented by an ordered list of the subintervals of $\mathcal{I}_B$, where, for for each subinterval, the corresponding $G_B(\lambda)$ and $C_B(\lambda)$ is recorded. The subintervals can be represented so as to allow $O(\log |L_B|) = O(\log r_{i+1})$ access time if, for example, we use balanced binary search trees (see [FeSl94] for one way of doing this). We should note that a binary search tree representation also allows efficient updates, a fact that shall be used by the search algorithm. Observe also that it is easy to build efficient representations for the r_{k+1}-regions at the beginning of iteration k: Since each region consists of a single vertex, properties (P1), (P2) and (P3) will trivially hold true at the beginning of iteration k if we set $L_B = \emptyset$ for every r_{k+1}-region B.

To lift iteration i of MST, we first build sets L_A, for all r_i-regions A, that satisfy properties (P1), (P2), and (P3) with respect to the r_i-regions. The whole process will be referred to as a *refinement* of $\mathcal{I}$. After the refinement is complete, we will process each r_i-region A separately, lifting the execution of REDUCE for each subinterval of the subdivision $\mathcal{I}_A$ of $\mathcal{I}$ induced by the points in L_A. Initially, for each r_i-region A, $L_A = \bigcup\{L_B : B \text{ is an } r_{i+1}\text{-subregion of } A\}$; this set will clearly satisfy property (P1) with respect to

region A. Property (P2) implies that the (reduced) graph $G[A]$ processed by REDUCE within any subinterval $\mathcal{I}'$ of $\mathcal{I}_A$ is the same for any $\lambda \in \mathcal{I}'$. The reason is that each subregion of A is reduced to a unique graph (indeed, a unique forest) within $\mathcal{I}'$. However, REDUCE's computation on A may have different outcomes for different $\lambda \in \mathcal{I}'$ and, thus, L_A might not satisfy (P2) with respect to A. Intuitively, this is because edges in different subregions of A may interact in ways that are not reflected by the current L_A — this occurs because cycles are formed when reduced subregions are put together (see Figure 1). Thus, it will be necessary to add new points to the L_A's corresponding to the intersections of cost lines of edges lying in different r_{i+1}-subregions of A. This will have to be done carefully, since the number of intersections may be large. Our technique is to sample the set of intersection points, and use the information, in conjunction with the oracle, to eliminate a large enough fraction of these points from any further consideration, without actually having to generate them.

A top-level description of the search algorithm is shown below.

```
SEARCH(G)
1   I ← (−∞, +∞)
2   build a multilevel division of G
3   for i ← k downto 1 do
4       refine I
5       for each r_i-region A do
6           for each interval I' of I_A do
7               lift REDUCE(A, C_A(λ)) over all λ ∈ I'
8               C_A(λ) ← C_A(λ) + Σ{C_B(λ) : B is an r_{i+1}-subregion of A}
9   construct a description of Z(λ) within I and locate λ*
```

Steps 1–2 are independent of edge costs and need not be explained further. In the rest of this section, we shall describe, in order, each of the main parts of SEARCH: refining (step 4), lifting (steps 5–8), and concluding the search (step 9). Additionally, we will discuss how the results of the lifting steps are used to produce faster oracles.

4.2 Refining the search interval

Consider any r_i-region A at the beginning of iteration i and let $\mathcal{I}'$ be any subinterval of $\mathcal{I}_A$. We write $S_A(\mathcal{I}')$ to denote the set of cost lines of edges

in $E(G_A(\lambda))$, for $\lambda \in \mathcal{I}'$. Because of property (P2), this is a well-defined set. The interval refinement algorithm uses a simple result. In its proof, for clarity, we will use a superscript of i to denote the value of an object at the beginning of iteration i; thus, for example, $G_B^i(\lambda)$ will be the reduced graph at the beginning of iteration i for the given λ-value.

Lemma 4.1 *Let $\mathcal{I}'$ be any subinterval of $\mathcal{I}_A$ at the beginning of iteration i of* SEARCH. *Let L' be the set of λ-values of all intersections of lines in $S_A(\mathcal{I}')$ whose λ-coordinate fall in $\mathcal{I}'$ and let $\mathcal{I}''$ be any of the subintervals of $\mathcal{I}'$ induced by the values in L'. Then, when applied to region A,* REDUCE *follows the same computation path for all $\lambda \in \mathcal{I}''$.*

Proof. REDUCE's choice of edges to delete and edges to contract depends on the topology of $G_A^i(\lambda)$ — which, by (P2), is fixed with $\mathcal{I}'$ — and the relative ordering of the costs of the edges in $G_A^i(\lambda)$. By definition of $\mathcal{I}''$, this ordering is fixed within $\mathcal{I}''$. The lemma follows. □

Suppose we have an oracle; i.e., a procedure that, given a value λ_0, determines whether $\lambda_0 < \lambda^*$, $\lambda_0 = \lambda^*$, or $\lambda_0 > \lambda^*$. After applying the oracle to λ_0, this value will be said to be *resolved*. Lemma 4.1 could then be used to refine $\mathcal{I}$ by simply generating intersection points between lines in $S_A(\mathcal{I}')$ and then invoking the oracle repeatedly. There are at least two obstacles to overcome. First, the total number of intersection points over all r_i-regions is superlinear. Second, we must have a fast (indeed, sublinear) oracle to resolve λ-values. The first problem will be addressed by narrowing the search interval using global information, prior to actually generating any intersection points. To address the second difficulty, we use a *sequence* of successively faster oracles $\text{ORACLE}_k, \ldots, \text{ORACLE}_1$, whose details will be supplied later. For now, we limit ourselves to stating that ORACLE_i, the oracle for iteration i, can resolve any value λ_0 in $O(n/\sqrt{r_{i+1}})$ time.

The refinement step uses two subroutines. The first of these applies an oracle to narrow the search interval $\mathcal{I}$:

NARROW($\mathcal{I}, L, s,$ ORACLE): Given an interval $\mathcal{I}$ where $\lambda^* \in \mathcal{I}$, a list of values $L \subseteq \mathcal{I}$, a number s, and an oracle ORACLE, return an interval $\mathcal{I}' \subseteq \mathcal{I}$ such that $\lambda^* \in \mathcal{I}'$ and $|L \cap \mathcal{I}'| \leq s$, and discard every point in L lying outside of $\mathcal{I}'$.

NARROW is implemented using a standard technique (see, e.g., [Meg83]): Choose a median element of L and apply ORACLE to it; depending on the outcome of the call, either resolve all elements of L larger than the median or all elements smaller than the median. In either case, at least half of the elements in L will be resolved; these values can therefore be removed from further consideration and the interval $\mathcal{I}$ is updated accordingly. The process is repeated until $\mathcal{I}$ contains no more than s points of L.

Lemma 4.2 NARROW$(\mathcal{I}, L, s,$ ORACLE$)$ *runs in* $O(|L| + t \cdot \log(|L|/s))$ *time, where* t *is the running time of* ORACLE.

Proof. Since each oracle call reduces the number of points of L in $\mathcal{I}$ by at least half, after q calls, the number of points will be at most $|L|/2^q$. Hence, $q = \log(|L|/s)$ calls suffice to reduce $\mathcal{I}$ by the desired amount. The total time is therefore $O(|L| + t \cdot \log(|L|/s))$, where the $O(|L|)$ term accounts for the total overhead incurred in computing medians and the second term accounts for the total time spent by the oracle calls. □

The second subroutine allows us to sample the intersection points of an arrangement of lines without having to construct it explicitly.

SQRT-QUANTILES(S): Given a set of lines S, return the set of $\sqrt{s}$ quantiles of their intersection points, where $s = |S|$. That is, return a set of $\sqrt{s} - 1$ λ-values that split the set of intersection points into $\sqrt{s}$ equal-sized subsets (to within 1).

The subroutine SQRT-QUANTILES can be implemented using a procedure by Cole et al. [CSSS89], which, given s distinct lines, finds the intersection point with the kth smallest λ-coordinate in (optimal) $O(s \log s)$ time. The quantiles we need can be found with $O(\sqrt{s})$ calls to Cole et al.'s algorithm. The required time is $O(s^{3/2} \log s)$. Observe that the number of intersection points between consecutive quantiles is $O(s^{3/2})$.

There are two main phases in the process of refining interval $\mathcal{I}$. Phase 1 examines each region A and each subinterval of $\mathcal{I}_A$ and finds an evenly-distributed subset of the intersection points of the lines within the subinterval. It does not find *all* the intersection points because that would lead to a superlinear search algorithm. The intersection points that are generated will define smaller subintervals of $\mathcal{I}$ within which relatively few intersections

take place. Interval $\mathcal{I}$ is then narrowed so that the number of intersections within each region that fall within $\mathcal{I}$ is small. Phase 2 actually enumerates these intersections and then narrows $\mathcal{I}$ further. In addition to phases 1 and 2, there is a preliminary phase where certain data structures are set up.

REFINE
    ▷ Phase 0
1   **for each** r_i-region A of G **do**
2       $L_A \leftarrow \bigcup\{L_B : B$ is an r_{i+1}-region within $A\}$
3   build an efficient representation of $\mathcal{I}_A$
    ▷ Phase 1
4   **for each** r_i-region A of G **do**
5   **for each** subinterval $\mathcal{I}'$ of $\mathcal{I}_A$ **do**
6           $L_A \leftarrow L_A \cup (\text{SQRT-QUANTILES}(S_A(\mathcal{I}')) \cap \mathcal{I}')$
7   $L \leftarrow \bigcup\{L_A : A$ an r_i-region$\}$
8   NARROW$(\mathcal{I}, L, n/r_i, \text{ORACLE}_i)$
    ▷ Phase 2
9   **for each** r_i-region A of G **do**
10   **for each** subinterval $\mathcal{I}'$ of $\mathcal{I}_A$ **do**
11          $L_A \leftarrow L_A \cup \{\lambda \in \mathcal{I}' : \lambda$ is the intersection of two lines in $S_A(\mathcal{I}')\}$
12 $L \leftarrow \bigcup\{L_A : A$ an r_i-region$\}$
13 NARROW$(\mathcal{I}, L, n/r_i, \text{ORACLE}_i)$

Phase 0 is done by merging efficient representations for the $\mathcal{I}_B$'s of the r_{i+1}-regions. By (P3), the total number of intervals over all the resulting $\mathcal{I}_A$'s will be $O(n/r_{i+1})$, and, by (P1), any $\mathcal{I}_A$ will have $O(r_i^2)$ subintervals. The required $\mathcal{I}_A$'s can thus be assembled in $O((n/r_{i+1}) \log r_i)$ time.

Phases 1 and 2 rely on the efficient representations of the $\mathcal{I}_A$'s in order to retrieve the various $S_A(\mathcal{I}')$'s. By (P3), the total number of subintervals that will be considered over all executions of step 5 is $O(n/r_{i+1})$; i.e., the total number of times step 6 is executed is $O(n/r_{i+1})$. By Lemma 3.3, the graph $G_A(\lambda)$ for any such interval $\mathcal{I}'$ is of size $O(r_i/\sqrt{r_{i+1}})$; thus, in step 6, $|S_A(\mathcal{I}')| = O(r_i/\sqrt{r_{i+1}})$. Hence, each execution of step 6 takes $O((r_i^{3/2}/r_{i+1}^{3/4}) \log r_i)$ time and adds $O(r_i^{1/2}/r_{i+1}^{1/4})$ values to L_A. The total time spent over all executions of step 6 is therefore $O((nr_i^{3/2}/r_{i+1}^{7/4}) \log r_i)$ and at

step 7, $|L| = O(nr_i^{1/2}/r_{i+1}^{5/4})$. By Lemma 4.2, step 8 takes time

$$O\left(\frac{nr_i^{1/2}}{r_{i+1}^{5/4}} + \frac{n\log r_i}{\sqrt{r_{i+1}}}\right).$$

Hence, the total time for Phase 1 is

$$O\left(\frac{nr_i^{3/2}\log r_i}{r_{i+1}^{7/4}} + \frac{nr_i^{1/2}}{r_{i+1}^{5/4}} + \frac{n\log r_i}{\sqrt{r_{i+1}}}\right). \tag{1}$$

As a consequence of the call to NARROW in Phase 1, the total number of subintervals that will be considered over all executions of step 10 is $O(n/r_i)$; i.e., step 10 is executed $O(n/r_i)$ times. As in Phase 1, we will have $|S_A(\mathcal{I}')| = O(r_i/\sqrt{r_{i+1}})$ for any subinterval $\mathcal{I}'$ of $\mathcal{I}_A$. The definition of SQRT-QUANTILES ensures that the number of intersection points of lines in $S_A(\mathcal{I}')$ that fall within $\mathcal{I}'$ is $O(|S_A(\mathcal{I}')|^{3/2})$. Hence, each execution of step 11 adds $O(r_i^{3/2}/r_{i+1}^{3/4})$ values to L_A, implying that, in step 12

$$|L| = O\left(\frac{n}{r_i}\cdot\frac{r_i^{3/2}}{r_{i+1}^{3/4}}\right) = O\left(\frac{nr_i^{1/2}}{r_{i+1}^{3/4}}\right).$$

Using standard techniques [CLR90], we can generate all the new points in $\mathcal{I}'$ in $O(\log|S_A(\mathcal{I}')|)$ time per point, at the expense of $O(|S_A(\mathcal{I}')|\log|S_A(\mathcal{I}')|)$ preprocessing time, for a total running time of

$$O(|S_A(\mathcal{I}')|^{3/2}\log|S_A(\mathcal{I}')|) = O\left(\frac{r_i^{3/2}}{r_{i+1}^{3/4}}\right)$$

per $\mathcal{I}'$. The time required over all $O(n/r_i)$ intervals is $O((nr_i^{1/2}/r_{i+1}^{3/4})\log r_i)$. By Lemma 4.2, step 13 takes time

$$O\left(\frac{n\log r_i}{\sqrt{r_{i+1}}} + \frac{nr_i^{1/2}}{r_{i+1}^{3/4}}\right), \tag{2}$$

which is also an upper bound on the time needed by Phase 2.

We summarize the analysis of REFINE with the following lemma.

Lemma 4.3 *Algorithm* SEARCH *spends a total of* $O(n\sum_{i=1}^{k}(r_i^{3/2}/r_{i+1}^{7/4})\log r_i)$ *time on* REFINE. *After* REFINE *is executed in stage* i, $|\bigcup\{L_A : A \text{ an } r_i\text{-region}\}| = O(n/r_i)$ *and for any* r_i*-region* A *and any subinterval* $\mathcal{I}'$ *of* $\mathcal{I}_A$, *the computation path followed by* REDUCE *on* A *is the same for all* $\lambda \in \mathcal{I}'$.

Proof. The time bound follows from equations (1) and (2). By construction, no subinterval $\mathcal{I}'$ of $\mathcal{I}_A$ will contain an intersection point of two lines in $S_A(\mathcal{I}')$. The uniqueness of the computation path within $\mathcal{I}'$ follows from Lemma 4.1. □

4.3 Lifting the ith iteration

We refer the reader back to the top-level description of algorithm SEARCH. Consider any r_i-region A at the point in iteration i of SEARCH immediately following the refinement of $\mathcal{I}$. By Lemma 4.3, the graph $G_A(\lambda)$ is the same for all $\lambda \in \mathcal{I}'$ for any subinterval $\mathcal{I}'$ of $\mathcal{I}_A$. This makes lifting the computation of REDUCE over all $\lambda \in \mathcal{I}'$ easy: simply execute REDUCE on A for *any* λ_0 in the interior of $\mathcal{I}'$. By Lemma 3.3, we have $|V(G_A(\lambda))| = O(r_i/\sqrt{r_{i+1}})$. Thus, if Kruskal's algorithm is used to compute minimum spanning trees, REDUCE will take $O((r_i/\sqrt{r_{i+1}})\log r_i)$ time. By Lemma 4.3, the total number of intervals over which the computation will be lifted is $O(n/r_i)$; Thus, the total time required for lifting all executions of REDUCE on r_i-regions is $O(n/\sqrt{r_{i+1}})$.

After lifting the different executions of REDUCE over all r_i-regions, SEARCH will use the information gathered in this process to construct efficient representations of the $\mathcal{I}_A$'s for all r_i-regions A. If balanced binary search trees are used for this purpose, the time required will be $O(\log |L_A|)$ per value in L_A. Since every point in L_A is the intersection of the cost lines of two edges in region A, $|L_A| = O(r_i^2)$, and, since the total number of subintervals over all r_i-regions is $O(n/r_i)$, all the required efficient representations can be built in $O((n/r_i)\log r_i)$ time.

4.4 The oracles

Recall that every edge in the input graph has a linear weight function $w_e(\lambda) = a_e - \lambda b_e$. As a result, $Z(\lambda)$ is a piecewise-linear concave function [Meg79], and, assuming the b_e's are nonnegative, the slope of every segment of $Z(\lambda)$ will be nonpositive. We thus have three possibilities [Meg83]:

- $Z(\lambda_0) = 0$. Then, $\lambda_0 = \lambda^*$
- $Z(\lambda_0) > 0$. Then, $\lambda_0 < \lambda^*$.

- $Z(\lambda_0) < 0$. Then, $\lambda_0 > \lambda^*$.

Therefore, to implement an oracle, it is enough to have a way to evaluate $Z(\lambda_0)$, the cost of a minimum spanning tree in G relative to the weights $w(\lambda_0)$; given this information, the additional work is $O(1)$. In fact, we only need to be able to evaluate $Z(\lambda)$ for $\lambda \in \mathcal{I}$, since any value $\lambda_0 \notin \mathcal{I}$ can be resolved in $O(1)$ time by determining its position relative to the endpoints of $\mathcal{I}$.

ORACLE$_k$ uses the planar minimum spanning tree algorithm of Section 3 on the original graph and thus takes $O(n)$ time (alternatively, the Cheriton-Tarjan algorithm could be used). We use the representation of the $\mathcal{I}_A$'s constructed while lifting the executions of REDUCE to implement ORACLE$_{i-1}$. The idea is to consult the representation to retrieve the reduced graphs at λ, thereby avoiding the wasteful task of recomputing this information from scratch. The details are as follows.

ORACLE$_{i-1}(\lambda_0)$
1 **for each** r_i-region A **do**
2 retrieve $G_A^i(\lambda_0)$ and $C_A^i(\lambda_0)$
3 assemble the $G_A^i(\lambda_0)$'s into a graph G'
4 $Z \leftarrow \text{MST}(G') + \sum\{C_A^i(\lambda_0) : A \text{ an } r_i\text{-region}\}$
5 **if** $Z > 0$ **then return** "$\lambda_0 < \lambda^*$"
else if $Z < 0$ **then return** "$\lambda_0 > \lambda^*$"
else return "$\lambda_0 = \lambda^*$"

Step 2 is done in $O(n \log r_i/r_i)$ by accessing the efficient representations of the $\mathcal{I}_A$'s. Step 3 is done by identifying boundary vertices of the $G_A^i(\lambda_0)$'s. The resulting graph G' is planar and has size $O(n/\sqrt{r_i})$. Thus, step 4 takes $O(n/\sqrt{r_i})$ time and, by Lemma 3.2, Z is the cost of a minimum spanning tree in the original graph G. The value Z can be used to resolve λ_0 in $O(1)$ time. Therefore, the oracle takes $O(n/\sqrt{r_i})$ time.

4.5 Concluding the search

Up to now, we have not specified the values of the r_i's. Assume that we choose $r_1 = n$; this makes the top-level region A the entire graph G. Then, when SEARCH reaches step 9, L_A will contain $O(1)$ values that subdivide $\mathcal{I}_A$

into $O(1)$ subintervals. Within each subinterval $\mathcal{I}'$, G will have been reduced to a one-vertex graph, and we will have a linear function giving the cost of the minimum spanning tree for all $\lambda \in \mathcal{I}'$. We can locate the point at which $Z(\lambda)$ equals zero by examining every interval to determine whether its associated cost line intersects the λ-axis. Thus, the final step of the search takes only $O(1)$ time.

The total time taken by SEARCH is therefore dominated by interval refinement, which, by Lemma 4.3, is $O(n \sum_{i=1}^{k} (r_i^{3/2}/r_{i+1}^{7/4}) \log r_i)$. Making $r_1 = n$, $r_{i+1} = r_i/\beta$, and setting k such that $r_k \leq \beta$, for some suitable $\beta > 1$, we obtain our main result. (We assume, for simplicity, that the successive r_i's are integers. The analysis can be easily modified to handle the case where they are not.)

Theorem 4.4 *The minimum ratio spanning tree problem can be solved in linear time for planar graphs.*

Before concluding this section, we should note that Cole et al.'s algorithm [CSSS89], which is used extensively in SQRT-QUANTILES, uses the AKS sorting network, which makes it impractical. Cole et al. have described a much simpler algorithm whose running time is $O(s \log^4 s)$ when a $O(\log^2 s)$-depth sorter, such as Batcher's [Bat68], is used. It is easy to verify that by using the simpler $O(s \log^4 s)$ algorithm, one can obtain a new MRST algorithm whose running time is still linear.

5 Discussion

The MRST algorithm can easily be adapted to yield linear-time algorithms for two closely related problems. In what follows, it is not necessary to assume that the b_e's are all positive or all negative. The first problem, which arises in certain types of sensitivity analysis [Gus83], is finding the next breakpoint of $Z(\lambda)$; i.e., given a real number λ_1 find the smallest $\lambda^* > \lambda_1$ such that λ^* is a breakpoint of $Z(\lambda)$. The second problem, which arises in Lagrangian relaxation [CMV89], is to find a maximizer λ^* of $Z(\lambda)$; i.e., find a λ^* such that $Z(\lambda^*) = \max_\lambda Z(\lambda)$. The algorithms for these problems are nearly identical to the MRST algorithm, except for the oracle. For the first problem, given a value λ_0, the oracle must determine whether the solution that is optimal at λ_1 is the same as the one that is optimal at λ_0 [Gus83]. If so, $\lambda_0 \leq \lambda^*$;

otherwise, $\lambda_0 > \lambda^*$. For the problem of maximizing Z, $\lambda_0 \leq \lambda^*$ if the slope of Z at λ_0 is positive [CoMe93]. Thus, in both problems the situation is analogous to minimum ratio optimization: the oracle requires an evaluation of $Z(\lambda_0)$ plus additional work that is $O(1)$.

The maximization algorithm discussed above can be used to improve the efficiency of an algorithm for solving the Lagrangian dual of the minimum spanning tree problem with a fixed number of side constraints [AgFe92, CMV89]. The algorithm given in [AgFe92] specializes to a $O(n \log^d n)$ algorithm for planar graphs, where d is the number of side constraints; using the algorithm described here, we are able to reduce the running time to $O(n \log^{d-1} n)$. We note that the maximization algorithm also leads to a new linear-time algorithm for solving the planar minimum spanning problem with one degree constraint, using a formulation given by Ahuja, Magnanti, and Orlin [AMO92]. An algorithm for this problem on general graphs, which specializes to a linear-time algorithm for planar graphs, was given by Gabow and Tarjan [GaTa84].

Our algorithm works in linear time for any family of graphs that admit a linear-time decomposition into regions with a sublinear number of boundary vertices and where spanning trees can be computed in linear time. Using additional ideas [FeWi91], it also extends to other parametric problems associated with matroids on graphs whose circuits are defined to be subgraphs homeomorphic from some finite set of graphs [Mat79]. Another member of this family is the parametric minimum spanning pseudoforest problem [GaTa88].

References

[AgFe92] R. Agarwala and D. Fernández-Baca. Weighted multidimensional search and its application to convex optimization. DIMACS Technical Report 92-51, November, 1992. To appear in *SIAM J. Comput.*

[AKS83] M. Ajtai, J. Komlos, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19 (1983).

[AMO92] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Bat68] K.E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Summer Computer Conf.*, Vol. 32, pp. 307–314.

[CEGS92] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Diameter, width, closest line pair, and parametric searching. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry*, pp. 120– 129 (1992).

[CMV89] P.M. Camerini, F. Maffioli, and C. Vercellis. Multi-constrained matroidal knapsack problems. *Mathematical Programming* 45:211–231 (1989).

[CoMe93] E. Cohen and N. Megiddo. Maximizing concave functions in fixed dimension. In *Complexity in Numerical Computations*, P.M. Pardalos, ed., World Scientific Press 1993.

[Cole87] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. Assoc. Comput. Mach.*, 34(1):200–208, 1987.

[Cha77] R. Chandrasekaran. Minimal ratio spanning trees. *Networks*, 7: 335–342 (1977).

[ChTa76] D. Cheriton and R.E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742 (1976).

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[CSSS89] R. Cole, J.S. Salowe, W.L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Comput.*, 18:792–810 (1989).

[FeSl94] D. Fernández-Baca and G. Slutzki. Optimal parametric search on graphs of bounded tree-width. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, pp. 155–166, LNCS 824, Springer-Verlag, 1994.

[FeWi91] D. Fernández-Baca and M.A. Williams. On matroids and hierarchical graphs. *Information Processing Letters* **38** (1991), 117–121.

[Fre85b] G.N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.* 16:1004–1022 (1985).

[Fre90] G.N. Frederickson. Optimal algorithms for partitioning trees and locating p-centers in trees. Technical Report CSD-TR 1029, Department of Computer Science, Purdue University, October 1990.

[FrWi90] M. Fredman and D. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. 31st Annual IEEE Symp. on Foundations of Computer Science*, 1990, pp. 719–725.

[GaTa84] H.N. Gabow and R.E. Tarjan. Efficient algorithms for a family of matroid intersection problems. *J. Algorithms*, 5:80–131 (1984).

[GaTa88] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for finding a minimum spanning pseudoforest. *Information Processing Letters*, 27(5):259–263 (1988).

[GGST86] H.N. Gabow, Z. Galil, T. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122 (1986).

[Goo93] M.T. Goodrich. Planar separators and parallel polygon triangulation. *Manuscript.* A preliminary version appeared in *Proceedings of the 24th Annual Symposium on Theory of Computing*, 1992, pp. 507–516.

[Gus83] D. Gusfield. Parametric combinatorial computing and a problem in program module allocation. *J. Assoc. Comput. Mach.*, 30(3):551–563 (1983).

[KKT94] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Manuscript.* A preliminary version appeared in *STOC 94*.

[KRRS94] Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *Proc. STOC 94*.

[Kru56] J.B. Kruskal. On the shortest spanning tree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50 (1956).

[Len87] T. Lengauer. Efficient algorithms for finding minimum spanning forests in hierarchically defined graphs. *J. Algorithms*, 8:260–284 (1987).

[LiTa79] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math*, 36:177–189 (1979).

[MaSc93] J. Matoušek and O. Schwartzkopf. A deterministic algorithm for the three-dimensional diameter problem. In *Proceedings of 25th Annual Symposium on Theory of Computing*, pp. 478–484 (1993).

[Mat79] L.R. Matthews. Infinite subgraphs as matroid circuits. *Journal of Combinatorial Theory, Series B*, **27** (1979), 260–273.

[Meg79] N. Megiddo. Combinatorial optimization with rational objective functions. *Math. Oper. Res.*, 4:414–424 (1979).

[Meg83] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. Assoc. Comput. Mach.*, 30(4):852–865, 1983.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[Tol93a] S. Toledo. Maximizing non-linear convex functions in fixed dimension. In *Complexity in Numerical Computations*, P.M. Pardalos, ed., World Scientific Press 1993. A preliminary version appeared in FOCS 92.